Silhouette clipping

Silhouette Clipping

Pedro V. Sander Xianfeng Gu Steven J. Gortler Hugues Hoppe John Snyder
Harvard University Microsoft Research

Abstract

Approximating detailed models with coarse, texture-mapped meshes results in polygonal silhouettes. To eliminate this artifact, we introduce silhouette clipping, a framework for efficiently clipping the rendering of coarse geometry to the exact silhouette of the original model. The coarse mesh is obtained using progressive hulls, a novel representation with the nesting property required for proper clipping. We describe an improved technique for constructing texture and normal maps over this coarse mesh. Given a perspective view, silhouettes are efficiently extracted from the original mesh using a precomputed search tree. Within the tree, hierarchical culling is achieved using pairs of anchored cones. The extracted silhouette edges are used to set the hardware stencil buffer and alpha buffer, which in turn clip and antialias the rendered coarse geometry. Results demonstrate that silhouette clipping can produce renderings of similar quality to high-resolution meshes in less rendering time.

Keywords: Level of Detail Algorithms, Rendering Algorithms, Texture Mapping, Triangle Decimation.

1 Introduction

Rendering detailed surface models requires many triangles, resulting in a geometry processing bottleneck. Previous work shows that such models can be replaced with much coarser meshes by capturing the color and normal fields of the surface as texture maps and normal maps respectively [2, 3, 20, 26]. Although these techniques offer a good approximation, the coarse geometry betrays itself in the polygonal silhouette of the rendering. This is unfortunate since the silhouette is one of the strongest visual cues of the shape of an object [14], and moreover the complexity of the silhouette is often only $O(\sqrt{n})$ on the number n of faces in the original mesh.

In this paper, we introduce *silhouette clipping*, a framework for efficiently clipping the rendering of coarse geometry to the exact silhouette of the original model. As shown in Figure 1, our system performs the following steps.

Preprocess Given a dense original mesh:

- Build a *progressive hull* representation of the original mesh and extract from it a coarse mesh, which has the property that it encloses the original, allowing proper clipping (Section 3).
- Construct a texture map and/or normal map over each face of the coarse mesh by sampling the color and/or normal field of the original mesh (Section 4).
- Enter the edges of the original mesh into a search tree for efficient runtime extraction of silhouette edges (Section 5).

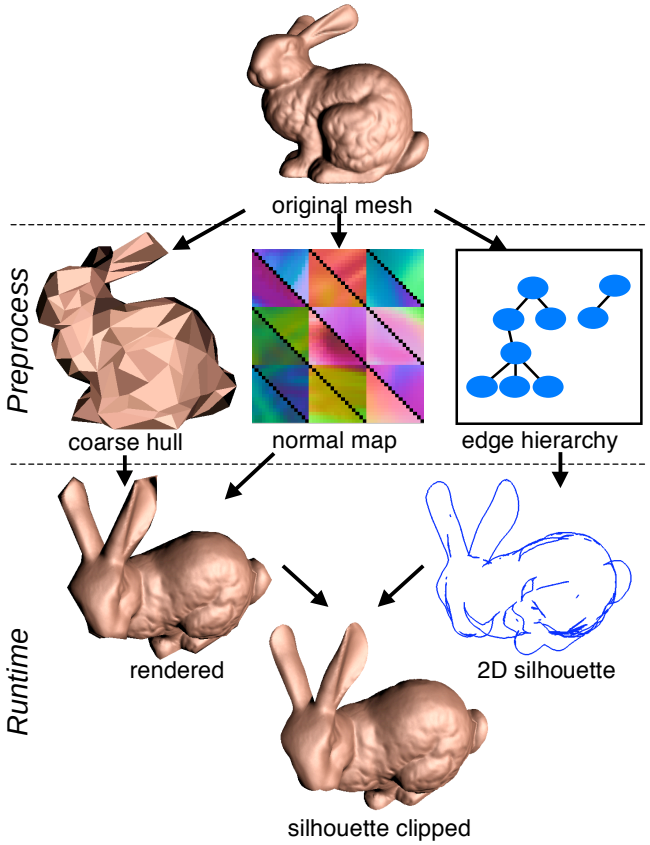


Figure 1: Overview of steps in silhouette clipping.

Runtime Then, for a given viewpoint:

- Extract the silhouette edges from the search tree (Section 5).
- Create a mask in the stencil buffer by drawing the silhouette edges as triangle fans. Optionally, draw the edges again as antialiased lines to set the alpha buffer (Section 6).
- Render the coarse mesh with its associated texture/normal maps, but clipped and antialiased using the stencil and alpha buffers.

Contributions This paper describes:

- The framework of silhouette clipping, whereby low-resolution geometry is rendered with a high-resolution silhouette.
- A progressive hull data structure for representing a nested sequence of approximating geometries. Within the sequence, any coarser mesh completely encloses any finer mesh.
- A new method for associating texel coordinates on the coarse model with positions on the original model. The association is based on the simple idea of shooting along an interpolated surface normal.
- A scheme for efficiently extracting the silhouette edges of a model under an arbitrary perspective view. It is inspired by previous work on backface culling [13, 15], but uses a convenient "anchored cone" primitive and a flexible n-ary tree to reduce extraction time.

http://cs.harvard.edu/~{pvs,xgu,sjg}
http://research.microsoft.com/~{hoppe,johnsny}

- An efficient technique for setting the stencil buffer given the silhouette edges. Special care is taken to overcome rasterization bottlenecks by reducing triangle eccentricities.
- An improvement for efficiently antialiasing the silhouette with little additional cost.
- Demonstrations that silhouette clipping produces renderings of similar quality to high-resolution meshes in less time.

Limitations

- Only the exterior silhouette is used for clipping and antialiasing. Internal silhouettes retain their polygonalized appearance from the coarse model.
- As in other texture mapping schemes, some minor texture slipping can occur, depending on the accuracy of the coarse model.
- Efficiency depends on a relative sparsity of silhouettes, and therefore breaks down for extremely rough geometry like trees or fractal mountains.
- The approach only works for static models represented by closed, oriented, 2-dimensional manifolds.
- The stencil setting method assumes that the viewpoint is outside the convex hull of the original model.

2 Previous Work

Level of Detail/Simplification Level-of-detail (LOD) techniques adapt mesh complexity to a changing view. The simplest approach precomputes a set of view-independent meshes at different resolutions from which an appropriate approximation is selected based on viewer distance (see survey in [10]). A more elaborate approach, termed view-dependent LOD [12, 18, 27], locally adapts the approximating mesh. Areas of the surface can be kept coarser if they are outside the view frustum, facing away from the viewer, or sufficiently far away. In particular, the view-dependent error metric of Hoppe [12] automatically refines near mesh silhouettes. However, a cascade of dependencies between refinement operations causes refinement in areas adjacent to the silhouette, increasing rendering load. Also, the efficiency of these systems relies on time-coherence of the viewing parameters.

With silhouette clipping, fewer polygons need to be processed since silhouettes are obtained as a 2D post-process. Antialiasing is achieved by processing only the silhouette edges rather than supersampling the entire frame buffer.

Texturing Maruya [20] and Soucy et al. [26] define textures over a coarse domain by following invertible mappings through a simplification process. The shape of the final parametrization is influenced by the fairly arbitrary sequence of simplification steps.

Cignoni et al. [2] describe a simple method for defining a parametrization using only the geometry of the coarse and fine models. Each position on the coarse model is associated with its closest point on the fine model. This method often creates mapping discontinuities in concave regions (Figure 4). In Section 4 we present a method that instead shoots rays along the interpolated surface normal. Although not guaranteed to produce a one-to-one mapping, our parametrization has far fewer discontinuities.

Silhouette Extraction Silhouette information has been used to enhance artistic renderings of 3D objects [6, 7, 19]. Blythe et al. [1] describe a multipass rendering algorithm to draw silhouettes in the screen. Other work highlights the visible silhouette by rendering thickened edges [24] or backfaces [23] translated slightly towards the viewpoint. These works require the traversal of the entire geometric object.

A number of algorithms exist for extracting silhouette edges from polyhedral models. Markosian et al. [19] describe a probabilistic algorithm that tests random subsets of edges and exploits view coherence to track contours. Their method is not guaranteed to find all of the silhouette components, and is too slow for models of high geometric complexity. Gooch et al. [7] extract silhouette edges efficiently using a hierarchical Gauss map. Their scheme is applicable only to orthographic views, whereas ours works for arbitrary perspective views.

Backface Culling Our method for fast silhouette extraction is inspired by previous schemes for fast backface culling. Kumar et al. [15] describe an exact test to verify that all faces are backfacing. They reduce its large cost by creating a memory-intensive auxiliary data structure that exploits frame-to-frame coherence. Johannsen and Carter [13] improve on this by introducing a conservative, constant-time backfacing test. The test is based on bounding the "backfacing viewpoint region" with a constant number of half spaces. In our system we use an even simpler anchored cone test primitive.

Johannsen and Carter do not address hierarchy construction, while Kumar et al. build their hierarchy using a dual space gridding that does not explicitly take into account the extraction cost. We describe a general bottom-up clustering strategy, similar to Huffman tree construction, that is greedy with respect to predicted extraction cost. In the results section we report the advantage of using our method over that of Johannsen and Carter.

Silhouette Mapping Our earlier system [8] performs silhouette clipping using an approximate silhouette, obtained using interpolation from a fixed number of precomputed silhouettes.

3 Progressive Hull

In order to be properly clipped by the high-resolution silhouette, the coarse mesh should completely enclose the original mesh M^n. In this section we show how such a coarse mesh can be obtained by representing M^n as a *progressive hull* — a sequence of nested approximating meshes $M^0 \ldots M^n$, such that

$$\mathcal{V}(M^0) \supseteq \mathcal{V}(M^1) \ldots \supseteq \mathcal{V}(M^n)$$

where $\mathcal{V}(M)$ denotes the set of points interior to M. A related construction for the special case of convex sets was explored in [4].

Interior volume To define interior volume, we assume that M^n is oriented and closed (i.e. it has no boundaries). In most cases, it is relatively clear which points lie in $\mathcal{V}(M)$. The definition of interior is less obvious in the presence of self-intersections, or when surfaces are nested (e.g. concentric spheres). To determine if a point $\mathbf{p} \in \mathbf{R}^3$ lies in $\mathcal{V}(M)$, select a ray from $\mathbf{p}$ off to infinity, and find all intersections of the ray with M. Assume without loss of generality that the ray intersects the mesh only within interiors of faces (i.e. not on any edges). Each intersection point is assigned a number, $+1$ or -1, equal to the sign of the dot product between the ray direction and the normal of the intersected face. Let the *winding number* $w_M(\mathbf{p})$ be the sum of these numbers [22]. Because the mesh is closed, it can be shown that $w_M(\mathbf{p})$ is independent of the chosen ray. To properly interact with the stencil algorithm described later in Section 6, we define interior volume using the *positive winding rule* as $\mathcal{V}(M) = \{\mathbf{p} \in \mathbf{R}^3 : w_M(\mathbf{p}) > 0\}$. Note that this description only defines interior volume; it is not used in actual processing.

Review of progressive mesh The progressive hull sequence is an adaptation of the earlier *progressive mesh* (PM) representation [11] developed for level-of-detail control and progressive transmission of geometry. The PM representation of a mesh M^n is obtained by simplifying the mesh through a sequence of n *edge collapse* transformations (Figure 3), thus defining a dense family of approximating meshes $M^0 \ldots M^n$.

For the purpose of level-of-detail control, edge collapses are selected so as to best preserve the appearance of the mesh during simplification (e.g. [3, 10, 11, 17]). We show that proper constraints on

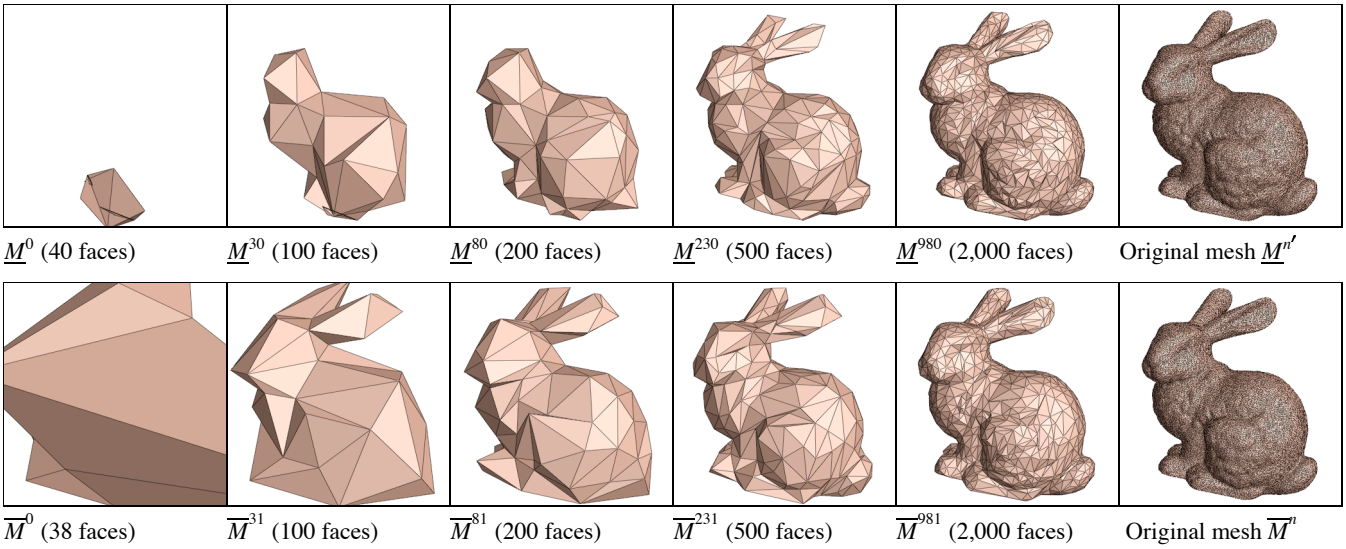


Figure 2: Example of progressive inner and outer hulls. The original mesh has 69,674 faces; n'=34,817; n=34,818.

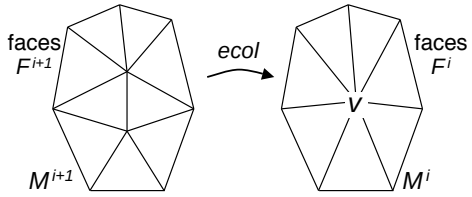


Figure 3: The edge collapse transformation.

the selection of edge collapse transformations allow the creation of PM sequences that are progressive hulls.

Progressive hull construction For the PM sequence to be a progressive hull, each edge collapse transformation $M^{i+1} \rightarrow M^i$ must satisfy the property $\mathcal{V}(M^i) \supseteq \mathcal{V}(M^{i+1})$. A sufficient condition is to guarantee that, at all points in space, the winding number either remains constant or increases:

$$\forall \mathbf{p} \in \mathbf{R}^3 \ , \ w_{M^{i+1}}(\mathbf{p}) \geq w_{M^i}(\mathbf{p}) \ .$$

Intuitively, the surface must either remain unchanged or locally move outwards everywhere.

Let F^{i+1} and F^i denote the sets of faces in the neighborhood of the edge collapse as shown in Figure 3, and let $\mathbf{v}$ be the position of the unified vertex in M^i. For each face $f \in F^{i+1}$, we constrain $\mathbf{v}$ to lie outside the plane containing face f. Note that the outside direction from a face is meaningful since the mesh is oriented. The resulting set of linear inequality constraints defines a feasible volume for the location of $\mathbf{v}$. The feasible volume may be empty, in which case the edge collapse transformation is disallowed. The transformation is also disallowed if either F^i or F^{i+1} contain self-intersections[1]. If $\mathbf{v}$ lies within the feasible volume, it can be shown that the faces F^i cannot intersect any of the faces F^{i+1}. Therefore, $F^i \cup \mathit{flip}(F^{i+1})$ forms a simply connected, non-intersecting, closed mesh enclosing the difference volume between M^i and M^{i+1}. The winding number $w(\mathbf{p})$ is increased by 1 within this difference volume and remains constant everywhere else. Therefore, $\mathcal{V}(M^i) \supseteq \mathcal{V}(M^{i+1})$.

The position $\mathbf{v}$ is found with a linear programming algorithm, using the above linear inequality constraints and the goal function of minimizing volume. Mesh volume, defined here as $\int_{\mathbf{p} \in \mathbf{R}^3} w_M(\mathbf{p}) d\mathbf{p}$, is a linear function on $\mathbf{v}$ that involves the ring of vertices adjacent to $\mathbf{v}$ (refer to [9, 17]).

As in earlier simplification schemes, candidate edge collapses are entered into a priority queue according to a cost metric. At each iteration, the edge with the lowest cost is collapsed, and the costs of affected edges are recomputed. Various cost metrics are possible. We obtain good results simply by minimizing the increase in volume, which matches the goal function used in positioning the vertex.

Inner and outer hulls The algorithm described so far constructs a *progressive outer hull* sequence $\overline{M}^0 \supseteq \ldots \supseteq \overline{M}^n$. By simply reversing the orientation of the initial mesh, the same construction gives rise to an *progressive inner hull* sequence $\underline{M}^0 \subseteq \ldots \subseteq \underline{M}^{n'}$. Combining these produces a single sequence of hulls

$$\underline{M}^0 \subseteq \ldots \subseteq \underline{M}^{n'} = \overline{M}^n \subseteq \ldots \subseteq \overline{M}^0$$

that bounds the original mesh from both sides, as shown in Figure 2. (Although the surface sometimes self-intersects, interior volume defined using the winding number rule is still correct.)

We expect that this representation will also find useful applications in the areas of occlusion detection and collision detection, particularly using a selective refinement framework [12, 27].

4 Texture Creation

As in [2, 20, 26], we create a texture tile over each face of the simplified mesh, and pack these tiles into a rectangular texture image. As illustrated in Figure 1, all tiles are right triangles of uniform size. The problem of texture creation is to fill these tiles using texel values (colors or normals) sampled from the original mesh. Inspired by Cignoni et al. [2], our approach constructs a parametrization from the simplified mesh to the original mesh based solely on their geometries (i.e. independent of the simplification algorithm). Whereas Cignoni et al. use a closest-point parametrization, we base our parametrization on a normal-shooting approach, which significantly reduces the number of discontinuities in the parametrization (see Figures 4 and 5).

Given the original mesh and a triangle T of the simplified mesh, we must determine how to calculate the values assigned to each texel. Our normal-shooting algorithm performs the following steps to compute the color or normal at each texel t of T:

[1] We currently hypothesize that preventing self-intersections in F^i and F^{i+1} may be unnecessary.

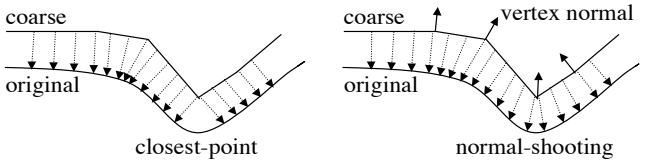
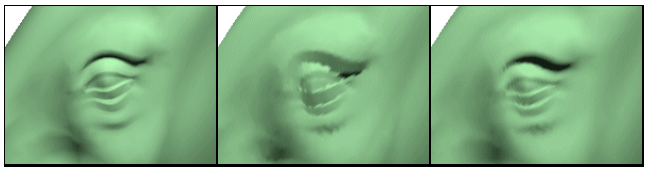


Figure 4: Closest-point parametrization often produces discontinuities not present with normal-shooting.



(a) original mesh (b) closest-point (c) normal-shooting

Figure 5: Comparison of texturing the coarse mesh using the closest-point parametrization and our normal-shooting parametrization. Note the parametric discontinuities in the concave regions for closest-point.

- Calculate the barycentric coordinates of t within the triangle T.
- Calculate the position $\mathbf{p}$ and normal $\vec{\mathbf{n}}$ by interpolating the positions and normals of the vertices of T.
- Shoot a ray from $\mathbf{p}$ in the $-\vec{\mathbf{n}}$ direction. This ray will intersect the original mesh at a particular point $\mathbf{q}$. In the extremely rare event of a ray failing to hit the original model, we instead use the closest point to $\mathbf{p}$.
- Given the triangle and barycentric coordinates of $\mathbf{q}$ in the original model, interpolate the prelit color or normal of its three vertices, and store the result in t.

We adjust the sampling resolution on the texture tiles depending on the complexities of the original and simplified meshes. For the models in Section 7, we sampled 512 texels per coarse face on the bunny and holes, but only 128 texels on the dragon, parasaur, and knot since these have many more coarse faces. These resolutions are enough to capture the desired level of detail. To allow bilinear interpolation on the resulting texture, we appropriately pad the triangle texture tiles.

5 Fast Silhouette Extraction

We consider each geometric edge in the mesh to consist of a pair of opposite-pointing *directed edges*. For a given mesh and viewpoint $\mathbf{p}$, the 3D silhouette is the subset of directed edges whose left adjacent face is frontfacing and whose right adjacent face is backfacing. More formally, a directed edge e is on the silhouette if and only if

$$\mathbf{p} \in \text{frontfacing}(e.f_1) \quad \text{and} \quad \mathbf{p} \notin \text{frontfacing}(e.f_2),$$

where the region

$$\text{frontfacing}(f) = \{\mathbf{p} \in \mathbf{R}^3 \mid (\mathbf{p} - f.\mathbf{v}) \cdot f.\vec{\mathbf{n}} \geq 0\}$$

in which $f.\mathbf{v}$ is any vertex of f, and $f.\vec{\mathbf{n}}$ is its outward facing normal.

Runtime Algorithm Applying this test to all edges in a brute-force manner proves to be too slow. Instead, our approach is to enter the edges into a hierarchical search tree, or more properly, a forest. Each node in the forest contains a (possibly empty) list of edges to test. Let the *face cluster* $F(n)$ for a node n be the set of faces attached to edges contained in that node and in all of its descendants. If for a given viewpoint we can determine that the faces in $F(n)$ are entirely frontfacing or entirely backfacing, then none of the edges contained in the node's subtree can be silhouettes, and thus the depth-first traversal skips the subtree below n. The basic structure of the algorithm is as follows:

```
procedure findSilhouetteEdges(node n, viewpoint p)
    if ( p ∈ frontfacing(F(n)) or p ∈ backfacing(F(n)) )
        return;    // skip this subtree
    for edges e in n.E
        if ( p ∈ frontfacing(e.f1) and p ∉ frontfacing(e.f2) )
            output(e);
    for children c in n.C
        findSilhouetteEdges(c,p);
```

The frontfacing and backfacing regions of a face cluster F are defined as

$$\text{frontfacing}(F) = \bigcap_{f \in F} \text{frontfacing}(f) \quad \text{and}$$

$$\text{backfacing}(F) = \bigcap_{f \in F} \overline{\text{frontfacing}(f)}\,.$$

To make hierarchical culling efficient, we need a fast, constant-time algorithm to conservatively test $\mathbf{p} \in \text{frontfacing}(F)$ and $\mathbf{p} \in \text{backfacing}(F)$. We do this by approximating these regions using two open-ended *anchored cones*, a_f and a_b, satisfying

$$a_f \subset \text{frontfacing}(F) \quad \text{and} \quad a_b \subset \text{backfacing}(F)$$

as shown in Figure 6. Each anchored cone a is specified by an anchor origin $a.\mathbf{o}$, normal $a.\vec{\mathbf{n}}$, and cone angle $a.\theta$. The construction of these cones will be presented shortly.

Each region test then reduces to

$$\mathbf{p} \in a \quad \Leftrightarrow \quad \cos^{-1}\left(\frac{\mathbf{p} - a.\mathbf{o}}{\|\mathbf{p} - a.\mathbf{o}\|} \cdot a.\vec{\mathbf{n}}\right) \leq a.\theta\,.$$

For efficiency and to reduce storage, we store in our data structure the scaled normal $a.\vec{\mathbf{n}}_s = a.\vec{\mathbf{n}} / \cos(a.\theta)$. With careful precomputation, the above test can be then implemented with two dot products and no square roots or trigonometric operations, via

$$\mathbf{p} \in a \quad \Leftrightarrow \quad (\mathbf{p} - a.\mathbf{o}) \cdot a.\vec{\mathbf{n}}_s \geq 0 \quad \text{and}$$
$$\left((\mathbf{p} - a.\mathbf{o}) \cdot a.\vec{\mathbf{n}}_s\right)^2 \geq \|\mathbf{p} - a.\mathbf{o}\|^2\,.$$

Because we construct a_f and a_b to have the same cone angle and opposite cone normals, we can test for inclusion in *both* anchored cones with just two dot products. This is made possible by precomputing and storing the "anchor separation" $d = (a_f.\mathbf{o} - a_b.\mathbf{o}) \cdot a_f.\vec{\mathbf{n}}$. For reference, the final node data structure is:

```
struct node
    vector scaledNormal;     //n_s
    point ffAnchor;          // a_f.o
    point bfAnchor;          // a_b.o
    float AnchorSeparation;  // d
    edgeList E;
    childPointerList C;
```

Anchored Cone Construction We first find the cone having the largest angle θ inside the frontfacing region. It can be shown that the central axis $\vec{\mathbf{n}}$ of such a cone has the following property: if one associates a point on the unit sphere with each face normal in the cluster, and computes the 3D convex hull of this pointset, $\vec{\mathbf{n}}$ must pass through the closest point from the origin to that convex hull. We therefore use Gilbert's algorithm [5] which directly finds this closest point in linear time. (Note that an open-ended cone exists

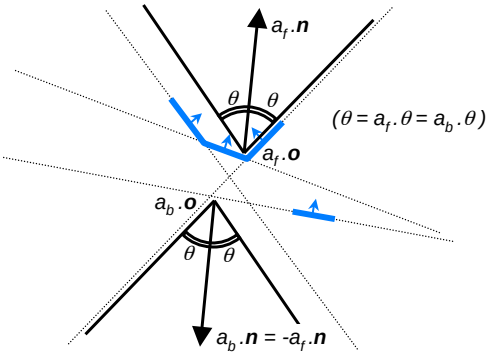


Figure 6: Anchored cones provide conservative bounds on the frontfacing and backfacing regions of a set of faces, illustrated here in 2D for the 4 oriented line segments in blue.

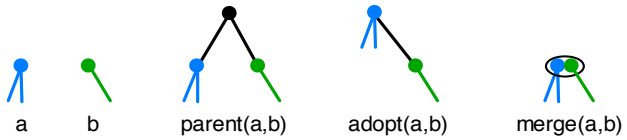


Figure 7: The three join operations.

if and only if the convex hull does not contain the origin.) The largest cone angle θ is then easily computed as the complement of the maximum angle from $\vec{\mathbf{n}}$ to the set of face normals. In fact, θ is also the complement of the angular distance of $\vec{\mathbf{n}}$ to any vertex in the closest simplex found by Gilbert's algorithm.

For a given node, we assign

$$\begin{aligned} a_f.\vec{\mathbf{n}} &= -a_b.\vec{\mathbf{n}} = \vec{\mathbf{n}} \\ a_f.\theta &= a_b.\theta = \theta \ . \end{aligned}$$

We then find the best cone origins, $a_f.\mathbf{o}$ and $a_b.\mathbf{o}$, by solving the linear programs

$$a_f.\mathbf{o} = \operatorname*{argmin}_{\mathbf{o} \in \text{frontfacing}(F)} \vec{\mathbf{n}} \cdot \mathbf{o} \quad \text{and} \quad a_b.\mathbf{o} = \operatorname*{argmin}_{\mathbf{o} \in \text{backfacing}(F)} -\vec{\mathbf{n}} \cdot \mathbf{o} \ .$$

Tree Construction We construct our trees in a bottom-up greedy fashion much like the construction of Huffman trees. We begin with a forest where each edge is in its own node. Given any two nodes (a,b), we allow the following three join operations (see Figure 7).

- parent(a,b): creates a new node with two children a and b.
- adopt(a,b): gives node b to node a as an additional child node.
- merge(a,b): creates a new node whose edge and child lists are the union of those from a and b.

Given these possible join operations, the algorithm is as follows:

```
Forest buildOptimalForest(Forest forest)
    candidates = buildJoinCandidates(forest);
    candidates.heapify();
    while (joinOp = candidates.removeTop())
        if ( joinOp.cost > 0 ) break;
        forest.applyJoin(joinOp);
        candidates.updateCosts(joinOp);
    return forest;
```

Candidate join operations are ordered in the heap by their predicted decrease in silhouette extraction cost. The silhouette extraction cost is computed as follows.

The cost of a forest is simply the sum of the costs of its roots:

$$\text{forestCost} = \sum_r \text{rootCost}(r) \ .$$

The cost of a root node is some constant k_a for the anchored cone tests, plus the possible cost of testing its edges and its children:

$$\text{rootCost}(r) = k_a + P(r) \left(k_e \left| r.E \right| + \sum_{c \in r.C} \text{nodeCost}(c, \{r\}) \right) \ ,$$

where k_e is the cost for testing an edge, and $P(r)$ is the probability of the node r not being culled[2]. To compute $P(r)$, one must assume some probability distribution over the viewpoints. We assume a uniform distribution over a large sphere U, in which case

$$P(r) = \frac{vol(U - r.a_f - r.a_b)}{vol(U)} \ .$$

The cost of a non-root node n with ancestor set A is computed recursively as:

$$\text{nodeCost}(n, A) = k_a + P(n \mid A) \left(k_e \left| n.E \right| + \sum_{c \in n.C} \text{nodeCost}(c, \{n\} \cup A) \right)$$

where $P(n \mid A)$ is the probability of the node n not being culled given that its ancestors A were also not culled. If one assumes that both anchored cones of a child are contained in its parent's, then

$$P(n \mid A) = \frac{vol(U - n.a_f - n.a_b)}{vol(U - p.a_f - p.a_b)}$$

where p is n's immediate parent. While this containment must be true of a node's respective frontfacing and backfacing regions, it is not necessarily true for their approximating anchored cones. In practice, numerical experiments have shown this approximation to be reasonable.

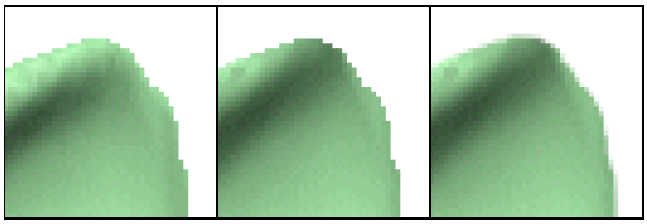
In principle one might consider all n^2 pairs of forest roots for candidate join operations. For computational efficiency during the preprocess, we limit the candidate set in the following way. A candidate graph is initialized with a graph vertex for each root in the initial forest, each representing a single mesh edge. Two vertices in the graph are linked if their corresponding mesh edges share the same mesh vertex, or if adjacent mesh faces have normals within an angular threshold[3]. Then during tree construction, when two roots are joined, their vertices and links are merged in the candidate graph.

6 Stencil Setting

The 3D silhouette extracted in the previous section is a set of directed edges. Since the mesh is closed and the silhouette edges separate frontfacing triangles from backfacing ones, the number of silhouette edges adjacent to any vertex must be even. Therefore the edges can be organized (non-uniquely) into a set of closed contours. Each such contour projects into the image plane as an oriented 2D polygon, possibly with many loops, and possibly self-intersecting. The winding number of this polygon at a 2D image location corresponds to the number of frontfacing surface layers that are seen along the ray from the viewpoint through that image location [19]. Our approach is to accumulate these winding numbers in the hardware *stencil buffer* for all contours in the 3D silhouette. Then, we clip the coarse geometry to the external silhouette of the original geometry by only rendering the coarse model where the stencil buffer values are positive.

[2] We have found that setting $k_a/k_e = 4/3$ gives us the best results.

[3] In practice we have found that ignoring similarity of normals (i.e., only considering mesh proximity) still provides search trees that are almost as good, with far less preprocessing time.



(a) original mesh (b) binary stencil (c) alpha matte

Figure 8: Comparison of rendering the bunny ear using the original mesh (69,674 face model), and using a coarse hull (500 face model) whose silhouette is (b) clipped to the stencil buffer and (c) antialiased using the alpha buffer.

Basic Algorithm The directed silhouette edges are organized into closed contours using a hash table. (For each directed edge, the hash key is the vertex index of the source vertex.) In order to render the winding number of each contour into the stencil buffer, we use a variation of the standard stencil algorithm for filling concave polygons [21]. Each edge contour is drawn as a fan of triangles about an arbitrary center point, which we choose to be the 3D centroid of the contour vertices. The orientation of each triangle determines whether its rendered pixels increment or decrement the stencil buffer values. To avoid testing triangle orientations in the CPU, we instead render the triangles twice, first with backface culling and stencil mode set to increment, and then with frontface culling and stencil mode set to decrement, as shown in the pseudocode below. The triangles are drawn as triangle fans for efficiency.

```
procedure setStencil(contours C, viewpoint q)
    setStencilToZero(boundingBox(C));
    cullFace(BACKFACE);
    for contours c in C
        point q = centroid(c.E);
        for edges e in c.E
            triangle t = makeTriangle(q, e.v1, e.v2);
            rasterizeToStencil(t, INCREMENT);
    cullFace(FRONTFACE);
    for contours c in C
        point q = centroid(c.E);
        for edges e in c.E
            triangle t = makeTriangle(q, e.v1, e.v2);
            rasterizeToStencil(t, DECREMENT);
    setDrawingToPositiveStencil();
```

Although the graphics hardware clips triangle fans to the view frustum, the setStencil algorithm remains correct even if parts of the model lie behind the viewer, as long as the viewer remains outside the convex hull of the object. This can be tracked efficiently by the test used in [25].

Loop Decomposition The basic algorithm described so far tends to draw many long, thin triangles. On many rasterizing chips (e.g. NVIDIA's TNT2), there is a large penalty for rendering such eccentric triangles. It is easy to show that the setStencil algorithm behaves best when the screen-space projection of **q** has a y coordinate at the median of the contour vertices. Choosing **q** as the 3D centroid of the contour vertices serves as a fast approximation.

To further reduce the eccentricity of the fan triangles, we break up each large contour into a set of smaller loops. More precisely, we pick two vertices on the contour, add to the data structure two opposing directed edges between these vertices, and proceed as before on the smaller loops thus formed.

When tested with the NVIDIA's TNT2, loop decomposition gave speedups of up to a factor of 2.3 on models that are raster bound on the stencil setting stage.

Model	Bunny	Dragon	Parasaur	Knot	Holes3
	Model complexities (number of faces)				
Original mesh	69,674	400,000	43,886	185,856	188,416
Coarse hull	500	4,000	1,020	928	500
	System timings (milliseconds)				
Original rendering	34.7	204.7	20.63	81.12	90.3
Silhouette extraction	4.5	24.2	4.0	6.5	4.0
Stencil setting	2.7	21.5	2.0	2.8	1.0
Coarse rendering	4.8	5.2	4.9	4.9	4.4
Total*	7.8	50.3	6.9	10.3	5.5
Speedup factor	4.4	4.1	3.0	7.9	16.4
(Antialiasing)	+3.0	+22.5	+2.9	+3.4	+1.5

Table 1: Timings of steps in our silhouette clipping scheme, and comparison with rendering the original mesh. *Total frame times are less than the sum due to parallelism between CPU and graphics.

Model	Bunny	Dragon	Parasaur	Knot	Holes3
Total faces	69,674	400,000	43,866	185,856	188,416
Total edges	104,511	600,000	65,799	278,784	282,624
	Silhouette extraction statistics				
Silhouette edges	3,461	23,493	3,227	3,291	1,737
Tested edges	10,256	67,934	10,938	13,134	5,976
Tested nodes	4,282	26,291	3,538	7,926	4,594
	Silhouette extraction times (milliseconds)				
Our search tree	4.1	28.2	4.3	6.4	3.3
Brute-force	20.4	117.3	12.5	50.6	51.4
Speedup factor	5.0	4.2	2.9	7.9	15.6

Table 2: Statistics of our silhouette extraction algorithm.

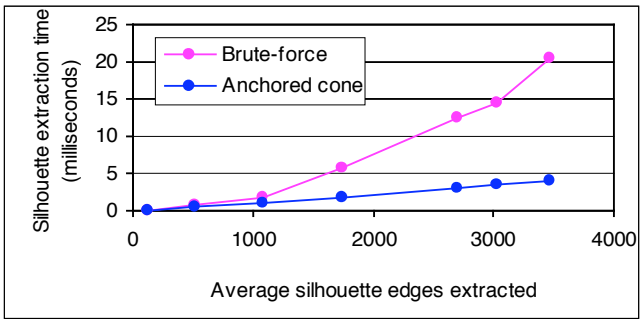
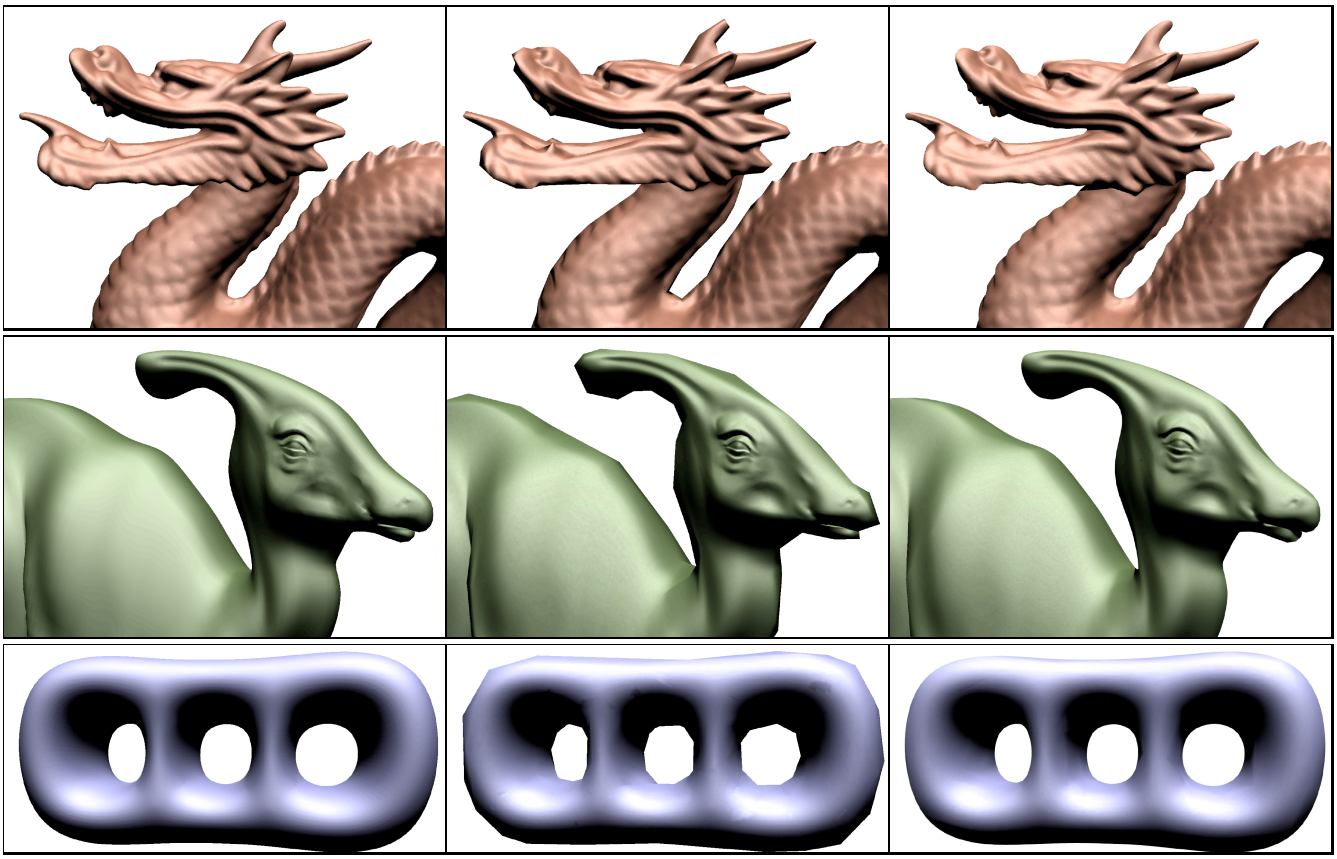


Figure 9: Comparison of the average silhouette extraction time with our algorithm and the brute-force algorithm, using bunny approximations with 500, 4,000, 20,000, 50,000, and 69,674 faces.

Antialiasing Although many graphics systems can antialias line segments, triangle antialiasing requires framebuffer supersampling which slows rendering except on high-end workstations. As a result, the silhouette typically suffers from aliasing artifacts (Figure 8a). The stencil buffer algorithm described in the previous section creates a *binary* pixel mask, therefore the coarse mesh clipped to this mask exhibits the same staircase artifacts (Figure 8b).

We can antialias the silhouette by applying line antialiasing on the silhouette contour. First, the silhouette edges are rendered as antialiased line segments into the alpha buffer (using glBlend(GL_ONE,GL_ZERO)). Second, the stencil buffer is computed as in the previous section. This binary stencil is then transferred to the alpha buffer, i.e. pixels interior to the silhouette are assigned alpha values of 1. Finally, the low-resolution geometry is rendered with these alpha buffer values using the *over* operation (glBlend(GL_DST_ALPHA,GL_ONE_MINUS_DST_ALPHA)). The result is shown in Figure 8c. As the timings in Table 1 reveal, silhouette antialiasing adds little to the overall time. Note that antialiased silhouette clipping on multiple models involves the non-commutative *over* operation, and thus requires visibility sorting [25].



(a) original mesh (b) simplified mesh (not hull) (c) silhouette-clipped coarse hull

Figure 10: Comparison of rendering the original mesh, a normal-mapped simplified mesh without the progressive hull constraints, and a coarse hull with the same number of faces but with silhouette clipping.

7 Results

We tested our framework on the five models of Table 1. The bunny and dragon are from 3D scans at Stanford University. (The dragon was simplified to 400,000 faces; the four boundaries in the base of the bunny were closed.) The parasaur is from the Viewpoint library. The 3-holed torus and knot are subdivision surfaces tessellated finely to obtain an accurate silhouette. We used normal maps for all of our examples.

Preprocessing a model consists of building a coarse hull, the normal and/or texture map, and the edge search structures. This takes between 30 minutes and 5 hours depending on model complexity.

We have focused our effort on optimizing the runtime algorithm. Times for the substeps of our scheme are shown in Table 1. These are obtained on a PC with a 550MHz Pentium III and a Creative Labs Annihilator 256 graphics card based on the NVIDIA GeForce 256 GPU. The execution times represent averages over many random views of the models. Note that the expense of extracting silhouette edges is significantly reduced due to parallelism between the CPU and GPU. For instance, silhouette extraction is nearly free for the bunny. We compare our approach of silhouette-clipping a coarse hull with rendering the original mesh, and find speedups of approximately 3 to 16. For rendering both the coarse hulls and the original meshes, we use precomputed triangle strips.

Figure 10 compares the image quality of the silhouette-clipped coarse hull with a simplified mesh of the same complexity and the original mesh. Figure 11 indicates that given a fixed amount of resources, our system can render a model with a silhouette of much higher resolution than the brute-force method.

As shown in Table 2, our hierarchical culling scheme results in explicit silhouette testing of only a small fraction of the edges, particularly on the smooth models. In all cases, our extraction time is much lower than the brute-force approach of explicitly testing all edges. It works much like a quadtree search algorithm, which can find all cells that touch a line in $O(\sqrt{n})$ time. Figure 9 shows this comparison as a function of silhouette complexity for several simplified bunny meshes. The graph indicates that the time for our algorithm increases linearly on the number m of silhouette edges in the model, whereas the brute-force time increases linearly on the total number n of edges, which in this case is quadratic on m.

We implemented Johannsen and Carter's backface culling algorithm and modified it to extract silhouettes, in order to compare it with our silhouette extraction scheme. For this comparison we measured computation based on the number of edges explicitly tested and nodes traversed. We did not use wall-clock time because our implementation of Johannsen and Carter was not overly optimized. For bunnies with 500, 4000, 20,000, 50,000, and 69,674 faces, our speedup factors were 1.1, 1.3, 1.5, 2.0, and 2.1, respectively.

8 Summary and Future Work

We have shown that silhouette clipping is a practical framework for rendering simplified geometry while preserving the original model silhouette. The operations of extracting silhouette edges and setting the stencil buffer can be implemented efficiently at runtime. With little added cost, silhouette clipping also permits antialiasing of the silhouette, a feature previously available only through expensive supersampling. Several areas for future work remain.

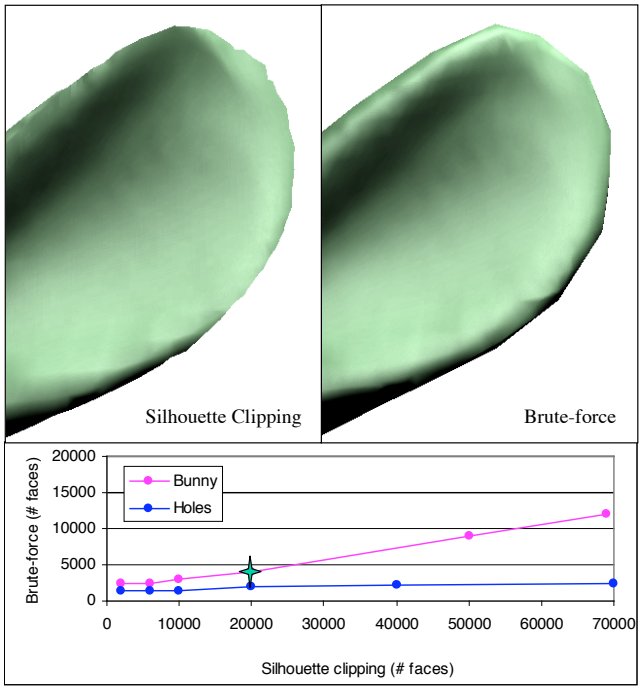


Figure 11: Comparison between silhouette clipping and brute-force rendering. The x-axis represents the resolution of the model used for silhouette extraction. The resolution of the coarse hull was fixed at 500 faces. The curves represent configurations that take the same amount of time to render. The star represents the configuration used in the bunny ear example shown above.

The complexity of the extracted silhouette should be adapted to the view, since it is obviously unnecessary to extract thousands of edges from an object covering a few pixels. Given a set of LOD meshes, our framework can use these for silhouette extraction by creating for each one a corresponding coarser hull. Alternatively, all of the silhouette meshes and their associated coarse hulls could be extracted from a single progressive hull. A related idea is to perform higher-order interpolation on the silhouette using projected derivatives or curvatures in addition to 2D points. This would result in smoother silhouettes without extracting more silhouette edges.

Currently, silhouette clipping only improves the appearance of exterior silhouettes. We have considered several approaches for dealing with interior silhouettes. One possibility is to exploit the winding number computed in the stencil buffer. Another approach partitions the mesh and applies silhouette clipping to each piece independently. We have performed initial experiments along these lines, but have not yet obtained a satisfactory solution.

Since the exterior silhouette of a shape is determined by its visual hull [16], silhouette extraction is unaffected by any simplification of the original mesh that preserves its visual hull. As an example, the interior concavity of a bowl can be simplified until it spans the bowl's rim. Such simplification offers an opportunity for further reducing silhouette extraction cost.

Acknowledgments

For the models we would like to thank Stanford University and Viewpoint DataLabs. We would also like to thank the MIT Laboratory for Computer Science for use of their equipment. The first three authors have been supported in part by the NSF, Sloan Foundation, and Microsoft Research.

References

[1] BLYTHE, D., GRANTHAM, B., NELSON, S., AND MCREYNOLDS, T. Advanced graphics programming techniques using OpenGL. avail from www.opengl.org.

[2] CIGNONI, P., MONTANI, C., ROCCHINI, C., AND SCOPIGNO, R. A general method for preserving attribute values on simplified meshes. In *Visualization '98 Proceedings*, IEEE, pp. 59–66.

[3] COHEN, J., OLANO, M., AND MANOCHA, D. Appearance-preserving simplification. *SIGGRAPH '98*, 115–122.

[4] DOBKIN, D. P., AND KIRKPATRICK, D. Determining the separation of preprocessed polyhedra – a unified approach. *ICALP-90, LNCS 443* (1990), 400–413.

[5] GILBERT, E. G., JOHNSON, D., AND KEERTHI, S. A Fast Procedure for Computing the Distance Between Complex Objects in Three-Dimensional Space. *IEEE Journal Of Robotics and Automation*, 2 (April 1988), 193–203.

[6] GOOCH, A., GOOCH, B., SHIRLEY, P., AND COHEN, E. A Non-Photorealistic Lighting Model for Automatic Technical Illustration. *SIGGRAPH 98*, 447–452.

[7] GOOCH, B., SLOAN, P., GOOCH, A., SHIRLEY, P., AND RIESENFELD, R. Interactive Technical Illustration. *ACM Symposium on Interactive 3D graphics 1999*, 31–38.

[8] GU, X., GORTLER, S., HOPPE, H., MCMILLAN, L., BROWN, B., AND STONE, A. Silhouette Mapping. Technical Report TR-1-99, Department of Computer Science, Harvard University, March 1999.

[9] GUÉZIEC, A. Surface simplification with variable tolerance. In *Proceedings of the Second International Symposium on Medical Robotics and Computer Assisted Surgery* (November 1995), pp. 132–139.

[10] HECKBERT, P., AND GARLAND, M. Survey of polygonal surface simplification algorithms. In *Multiresolution surface modeling (SIGGRAPH '97 Course notes #25)*. ACM SIGGRAPH, 1997.

[11] HOPPE, H. Progressive meshes. *SIGGRAPH '96*, 99–108.

[12] HOPPE, H. View-dependent refinement of progressive meshes. *SIGGRAPH '97*, 189–198.

[13] JOHANNSEN, A., AND CARTER, M. B. Clustered Backface Culling. *Journal of Graphics Tools 3*, 1 (1998), 1–14.

[14] KOENDERINK, J. J. What does the occluding contour tell us about solid shape. *Perception 13* (1984), 321–330.

[15] KUMAR, S., MANOCHA, D., GARRETT, W., AND LIN, M. Hierarchical Back-Face Computation. *Eurographics Rendering Workshop 1996*, 235–244.

[16] LAURENTINI, A. The visual hull concept for silhouette based image understanding. *IEEE PAMI 16*, 2 (1994), 150–162.

[17] LINDSTROM, P., AND TURK, G. Fast and memory efficient polygonal simplification. In *Visualization '98 Proceedings*, IEEE, pp. 279–286.

[18] LUEBKE, D., AND ERIKSON, C. View-dependent simplification of arbitrary polygonal environments. *SIGGRAPH '97*, 199–208.

[19] MARKOSIAN, L., KOWALSKI, M., TRYCHIN, S., AND HUGUES, J. Real time non photorealistic rendering. *SIGGRAPH '97*, 415–420.

[20] MARUYA, M. Generating texture map from object-surface texture data. *Computer Graphics Forum (Proceedings of Eurographics '95) 14*, 3 (1995), 397–405.

[21] NEIDER, J., DAVIS, T., AND WOO, M. *OpenGL Programming Guide, Second Edition*. Addison-Wesley, 1997.

[22] NEWELL, M. E., AND SEQUIN, C. The Inside Story on Self-Intersecting Polygons. *Lambda 1*, 2 (1980), 20–24.

[23] RASKAR, R., AND COHEN, M. Image Precision Silhouette Edges. *ACM Symposium on Interactive 3D Graphics 1999*, 135–140.

[24] ROSSIGNAC, J., AND VAN EMMERIK, M. Hidden contours on a frame-buffer. *Proceedings of the 7th Workshop on Computer Graphics Hardware* (1992).

[25] SNYDER, J., AND LENGYEL, J. Visibility Sorting and Compositing without Splitting for Image Layer Decompositions. *SIGGRAPH '98*, 219–230.

[26] SOUCY, M., GODIN, G., AND RIOUX, M. A texture-mapping approach for the compression of colored 3D triangulations. *The Visual Computer 12* (1986), 503–514.

[27] XIA, J., AND VARSHNEY, A. Dynamic view-dependent simplification for polygonal models. In *Visualization '96 Proceedings*, IEEE, pp. 327–334.